ELSEVIER

Theoretical Computer Science 245 (2000) 175–202

Theoretical Computer Science

www.elsevier.com/locate/tcs

Brigitte Oesterdiekhoff

Heinz Nixdorf Institute and Department of Mathematics & Computer Science, University of Paderborn, FB 17-Mathematik-Informatik, Postfach 1621, D-33095 Paderborn, Germany

Abstract

We survey recent results on periodic algorithms. We focus on the problems of sorting, merging and permuting and concentrate on algorithms that have small constant periods. © 2000 Elsevier Science B.V. All rights reserved.

Keywords: Periodic network; Compare–exchange operation; Comparator; Sorting; Merging; Permutation routing; Switch; Shift

1. Introduction

We consider algorithms of a special kind designed for simple networks consisting of weak processing units. We concentrate on one feature, namely periodicity, that makes such algorithms easy to implement. We consider the so-called comparator networks that are typically used for tasks such as sorting, merging of sorted sequences and selection. Below we recall two alternative definitions of comparator networks:

The model: Comparator networks are traditionally defined as consisting of n wires connected by comparators (see Fig. 1). The n input items move on the wires from left to right. At each time there is exactly one item on each wire. The items may be exchanged between the wires by comparators. A comparator $[i,j]$ originating at wire i pointing to wire j performs a compare-exchange operation in the following way: if the item on wire i is greater than the item on wire j, then the items are exchanged. The comparators in the comparator network are grouped into layers so that a single wire is connected to at most one comparator in a layer. The items traveling traveling through the wires arrive simultaneously at a given layer and in one *parallel step* all comparators of the layer are applied. A comparator network is called a *standard* comparator network [10, p. 236], if $i<j$ for any comparator $[i,j]$. Since every non-standard sorting network

☆ Partially supported by DFG-Sonderforschungsbereich 376 "Massive Parallelität", DFG Leibniz Grant Me872/6-1 and EU ESPRIT Long Term Research Project 20244 (ALCOM-IT).

E-mail address: brigitte@uni-paderborn.de (B. Oesterdiekhoff).

PII: S0304-3975(99)00281-9

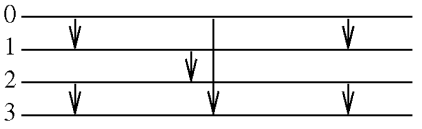


Fig. 1. A comparator network with 3 parallel steps.

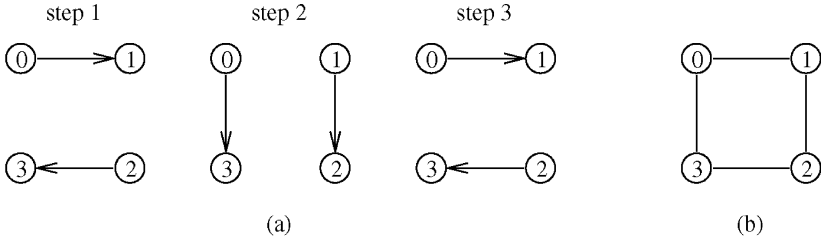


Fig. 2. (a) The algorithm of Fig. 1 in the graph model. (b) Its underlying graph.

can be converted easily into a standard one [10, p. 239], often only standard sorting networks are considered.

Alternatively, a comparator network can be defined in the *graph model*: Then each node of the graph stores exactly one item at a time. The items are exchanged between the nodes through comparators represented by directed edges between nodes. A comparator acts as before: the maximum of the values stored in the nodes adjacent to a comparator goes to the node pointed by the comparator, the minimum goes to the node where the comparator originates. During a single (parallel) step a set of comparators is applied (see Fig. 2(a)) with each node adjacent to at most one comparator. The graph consisting of the nodes and all edges used as comparators is called the *underlying graph* of the network (see Fig. 2(b)).

The difference between the two models reflect different implementation strategies. In the traditional model items are moved through the wires and sometimes exchanged by the comparators. This results in a constant degree network with only minor control logic. In the graph model the items are placed in the nodes. Communication links between the nodes may exchange the items. In this case usually more control logic must be realized in the nodes and the degree of the underlying graph depends on the sorting algorithm. However, we will see that the second model may become very attractive for periodic algorithms.

In the design and analysis of comparator networks the so-called 0–1-Principle [10, p. 224] turned out to be very useful. It says that if a comparator network sorts (merges) all inputs consisting solely of 0's and 1's, then it sorts (merges) arbitrary inputs. So it suffices to consider only sequences consisting of 0's and 1's. In this paper we will intensively use this principle.

Classical comparator networks: Odd–Even Transposition Sort (OETS) by Knuth [10, p. 241] is the simplest sorting comparator network (see Section 2). Its disadvantage is that for sorting n items it requires n parallel steps, so it is very slow.

The most famous comparator networks are Batcher's Bitonic Merge Sort and Odd–Even Merge Sort [2] and the AKS network [1]. Batcher's networks are elegant in design and need $\frac{1}{2}\log n(\log n + 1)$ parallel steps. The AKS network sorts in $c \log n$ steps. However c is a large constant. Paterson's [19] analysis of the AKS network results in $c \approx 6100$. Chvátal [4] shows that an improved construction leads to $c \approx 1830$. Batcher's networks have been used as a basis for practical implementations, regardless of the fact that the AKS network is asymptotically faster.

If the comparator networks mentioned in the paragraph before are laid out in the traditional model we have to use many layers and so a lot of hardware. If the algorithms are laid out in the graph model many communication links to one node are necessary, since the degrees of the underlying graphs of the AKS network and of Batcher's network are $\Theta(\log n)$ and $\log n$, respectively. A lot of control logic must be realized in the nodes in order to control the choice of compare–exchange operations to be executed. So the solution in the graph model is also expensive regarding hardware. (For the traditional model we may avoid waste of hardware through pipelining. In this case if the kth sequence is currently at layer i, then the sequence $k + j$ resides in layer $i - j$. However, this trick may be applied only if we have to sort many sequences!) Additionally, the AKS network is based on expanders which are very costly to be laid out.

Periodic comparator networks: In order to reduce the problems mentioned above we consider *periodic* comparator networks. A comparator network is called *k-periodic*, if for every moment t of the computation the parallel steps t and $t + k$ use the same comparators. k consecutive parallel steps are called a *round* of the algorithm.

If the period k of a comparator network is small, since we need hardware for k different steps only. For the traditional model there are two ways in which the hardware may become cheaper. First, it can be used repeatedly: The output produced by the network after k steps is fed again into the network as input by wrap-around edges (see Fig. 3). Alternatively, instead of wrap-around edges we can assemble the network from identical parts. In this case k steps are performed in one part and the output is moved to the next part for performing the following step. Since single parts have small size and since they are identical the circuits assembled are relatively simple and resistant to faults. Concerning the graph model, the underlying graph has a constant degree. Thus only a small number of communication links per node must be realized. The nodes might be quite primitive, since complicated control logic for determining operations to be executed is not needed.

In the graph and wrap-around mode the network can work in an adaptive mode, i.e. it can terminate execution if the items have already been sorted prior to the worst case runtime.

From the discussion above we see that periodic comparator networks might be efficient regarding hardware cost. Indeed, these costs heavily depend on the size of a single chip. It is evident that through an implementation with wrap-around edges we save a lot of space. A direct implementation as the graph model uses only a small number of communication links per processing unit, while the communication links

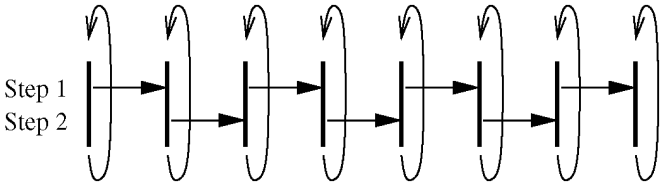


Fig. 3. "Wrap-around" description of OETS.

take usually most of the space on a chip. Even if we assemble sorting hardware from identical parts, as described in the paragraph above, we do not run into high costs, since single parts consisting of k parallel steps might be quite cheap. The gain is due to the fact that putting all steps on one chip is much more expensive, since the cost of a chip grows very fast with its size.

Nice implementation issues of periodic algorithms would have been useless, if the underlying algorithms were not time efficient. Hence we are faced with the following problem: *how to design time-efficient periodic algorithms.* One may claim that only for very few computational problems periodic algorithms may be as fast as their non-periodic counterparts. In this paper we present a series of results showing that this is not necessarily the case. We examine problems that at a first look have "non-periodic" nature, but have time-efficient periodic solutions.

A significant step in the direction of constructing periodic sorting algorithms is the construction of Dowd et al. [5], who propose the balanced sorting network. This network is $\log n$-periodic and sorts n items in time $\log^2 n$. The underlying graph of this network is the $\log n$-dimensional hypercube [6]. The strategy of the algorithm is to merge sorted subsequences until we get one sorted sequence. For this purpose the so-called balanced mergers are used. The point is that at every phase of the algorithm we use the same merging procedure no matter how big the sorted subsequences are. This distinguishes the balanced sorting network from Batcher's Bitonic Merge Sort and Odd–Even Merge Sort. Rudolph [20] shows that the balanced sorting network has good fault tolerance properties. In [3] the balanced sorting network is somewhat generalized.

Scherson et al. [23] and, independently, Sado and Igarashi [21] propose Shearsort algorithm for the two-dimensional $m \times \ell$ mesh architecture. This algorithm is $(m+\ell)$-periodic and performs $\lceil \log m \rceil + 1$ rounds in the worst case. A round consists of executing Odd–Even Transposition Sort on each row and, afterwards, on each column of the mesh. Kutyłowski and Wanka [15] generalize Shearsort to three-dimensional meshes. For the $\ell \times \ell \times \ell$ mesh, they show a $2\lceil \log \ell \rceil + 10$ upper bound on the number of rounds this 3ℓ-periodic algorithm has to perform, while $2\lfloor \log \ell \rfloor + 1$ is a lower bound.

Outline of the paper: In the rest of the paper we deal with constant-periodic networks. In Section 2 we discuss OETS. Section 3 contains a generalization of OETS on expanders and Section 4 contains generalizations on meshes. In Section 5 the strongest known result is discussed. There we describe a general method called the *periodification scheme* that converts automatically an arbitrary sorting network into a 3-periodic

sorting network with slight loss of efficiency. Section 6 presents time optimal merging networks of a constant period. Section 7 we discuss results on permutation networks using techniques related to the periodification scheme. We conclude with some remarks in Section 8.

2. Odd–Even Transposition Sort (OETS)

OETS [10, p. 241] is a 2-periodic sorting network that sorts n items in n steps. The first step of a round of OETS uses comparators from node (wire) i to node (wire) $i+1$, for all odd i. The second step of a round uses comparators from node (wire) i to node (wire) $i+1$, for all even i (see Fig. 3). The underlying graph of this algorithm is a linear array of n nodes.

It can easily be shown that any standard sorting network must contain each of the comparators used by OETS [10, p. 241]. Indeed, it suffices to consider an input containing 0's on positions 1 through $i-1$ and on position $i+1$, and containing 1's on the remaining positions. Only a comparator between node i and $i+1$ may move the displaced 0 to the right place! It follows that a single round of a periodic sorting network must contain all comparators of OETS. Another consequence is that any periodic sorting network requires at most n rounds [24].

During one step each node is adjacent to at most one comparator, so any 2-periodic standard sorting network is essentially OETS: one of its steps must contain all comparators of the first step of OETS, and the other step must contain all comparators of the second step of OETS. A similar argumentation works for merging networks, (at least if the non-decreasing sorted input sequences are allocated to nodes with increasing indices). We conclude that any 2-periodic network needs at least n steps for sorting and $n/2$ steps for merging. Therefore, the minimal period that is necessary for constructing sorting or merging networks with runtime $o(n)$ is at least 3.

Interleaving: OETS has one interesting property that will be discussed frequently in this paper. Namely, if we include additional layers to OETS, then we do not slow down the performance of OETS (and sometimes we accelerate it). More generally, we say that we *interleave* two networks N_1 and N_2, if we build a new network whose steps are the steps of N_1 interleaved with the steps of N_2. Generally, N_1 and N_2 may degrade the performance of each other, so finally the new network becomes worthless. But it is not the case for OETS [24].

3. Expander techniques

In the search of efficient fast constant-periodic networks Kik et al. [9] pursued an expander-based approach.

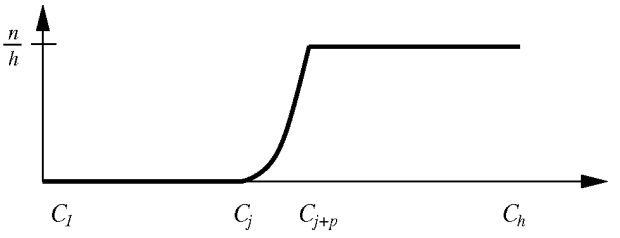


Fig. 4. The number of ones in the columns after O(h) steps of Column-OETS.

Theorem 1 (Kik et al. [9]). *For a fixed but arbitrary $k \in \mathbb{N}$, there is a comparator network of period* O(k) *that sorts n items in time* O($k^2 n^{1/k}$).

The network of Theorem 1 is a modification of OETS. The nodes of the network are organized in a mesh consisting of h columns $C_1, \ldots, C_h$ and therefore we call the algorithm Column-OETS. A single step of Column-OETS is analogous to a step of OETS on h nodes: instead of comparing nodes i and $i+1$, $i<h$, there are comparators between columns C_i and C_{i+1}, for all odd (even) i. Ideally, these columns should be sorted together, but this would require O($\log n/h$) steps. Instead we use the ε-halver of Ajtai et al. [1]. A network N is called a (V_1, V_2, ε)-halver on m elements if:

- The nodes of N are grouped in sets V_1, V_2, with $|V_1| = |V_2| = m/2$ and all comparators originate in V_1 and point to V_2;
- Let an input to N contain k ones and $m-k$ zeroes. If $k \leqslant m/2$, then after performing N at most $\varepsilon \cdot k$ ones are stored in V_1. If $k \geqslant m/2$, then the output of N contains at most $\varepsilon \cdot (m-k)$ zeroes stored in V_2.

It was shown, using expander graphs, that for each $\varepsilon > 0$ and $n \in \mathbb{N}$, there is an ε-halver for n inputs consisting of O($1/\varepsilon \cdot \log(1/\varepsilon)$) parallel steps [1].

By applying ε-halvers between C_i and C_{i+1} we achieve almost the same effect as by sorting them. Indeed, if after applying an ε-halver to C_i and C_{i+1}, t many ones are left in C_i, then there are at least $t \cdot (1/\varepsilon - 1)$ ones in C_{i+1}. Even for a moderately small $\varepsilon > 0$ the coefficient $1/\varepsilon - 1$ is large. If we assume even the most pessimistic situation that all ones are on a wrong side of the network, then nevertheless the ones are swept to the right side almost in the same manner as during OETS. One can prove that after O(h) steps of Column-OETS the ones are concentrated on the right side of the network: (see Fig. 4) for some $j \leqslant h$ the columns $C_1, \ldots, C_j$ contain only 0's, the columns $C_{j+p}, \ldots, C_n$ contain only 1's, where $p = \mathrm{O}(\log n)$. Within the columns C_{j+1}, …, C_{j+p-1} the upper bound on the number of 1's starts with a constant in C_{j+1} and then grows exponentially. In this way we achieve some state that may become stable while using only ε-halvers to push further the ones to the right-hand side.

At this point we use the interleaving technique to make our network sort precisely. A simple solution is to interleave the steps of Column-OETS with the genuine OETS on n nodes. In this special case the key point is that the comparators introduced do not disturb the work of the ε-halvers. For an ε-halver it is inessential what the input

looks like, the quality of the output is guaranteed by the number of ones. The same property holds for OETS as described in Section 2: including additional (standard) comparators does not increase the upper bound of the computation time. Therefore, we may think of our algorithm as consisting of two phases. During phase 1 the ones are moved to columns $C_{j+1},\ldots,C_n$ (for this part we analyze the ε-halvers and ignore OETS). During the second phase we sort the "dirty columns" $C_{j+1},\ldots C_{j+p-1}$ (using OETS and regardless of the ε-halvers).

In order to achieve the runtime $O(n^{1/k})$ we interleave more networks than proposed above. The idea is that for sorting the "dirty columns" we may use a network that is much faster than OETS. The only necessary condition is that this network must be immune to interleaving it with other comparators, that is, it should guarantee the same upper bound for computation time after inserting additional comparators. At this point we may use, for instance, OETS combined with Column-OETS for a smaller size of the columns. Then we may divide the computation into three (conceptual) phases:

Phase 1: We get a small number of dirty columns according to the first Column-OETS.

Phase 2: The region consisting of dirty columns is repartitioned into columns, but now the size of the columns is considerably smaller. The second Column-OETS is used to reduce the number of just created dirty columns.

Phase 3: The remaining dirty region is sorted by OETS.

4. Generalizations of OETS to meshes

Since OETS has bad runtime performance, due to the diameter of the underlying graph, one may try to speed it up by generalizing it for other graphs of a simple architecture. There have been many efforts to design such algorithms for meshes.

A d-dimensional $k_1\times k_2\times\cdots\times k_d$-mesh consists of $\prod_{i=1}^{d} k_i$ nodes. Each node corresponds to a d-dimensional vector $(i_1,i_2,\ldots i_d)$ where $1\leqslant i_\ell\leqslant k_\ell$ for $1\leqslant\ell\leqslant d$. Two nodes are linked by an edge, if they differ in precisely one coordinate and if the absolute value of the difference in that coordinate is 1. In the case $d=1$ the mesh is called a linear array of length k_1. In the special case that $k_i=2$ for $1\leqslant i\leqslant d$ the mesh is the d-dimensional hypercube.

Savari [22] proposed sorting algorithms generalizing OETS to meshes. She showed that for these (quite straightforward and natural) generalizations the runtime is $\Theta(n)$, even on average. So there is no significant improvement relative to the original OETS. She analyses among others a 4-periodic algorithm with a round consisting of steps shown in Fig. 5. The nodes of the network are arranged in an $N\times N$-mesh with the row-major ordering. The algorithm is formed by interleaving OETS (steps 2 and 4) with OETS performed separately on the columns (steps 1 and 3). Note that all wrap-around comparators between the last and the first column are necessary because all comparators between nodes i and $i+1$ must be included in any sorting network.

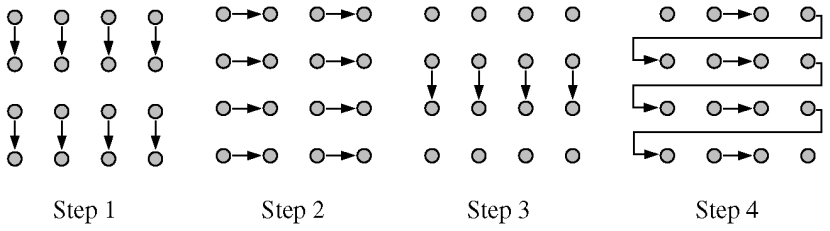


Fig. 5. The steps of one round of Savari's network.

The reason why this algorithm is so slow is the following phenomenon: Consider the input, where the nodes of the rightmost column contain N zeroes and the rest of the nodes contain 1's. The output for this case must contain all 0's in the lowest numbered row, i.e. at the top of the network. The 0's form a "tower" and travel together through the network from the right to the left until they reach the leftmost column. There the tower loses only one zero and the rest goes to the rightmost column through the wrap-around edges. Then everything starts from the beginning: the zeroes forming a tower go together to the left. Note that the tower loses one zero once every $\Theta(N)$ steps. Thus, we need $O(N^2)$ steps to reduce its height to 1 and the runtime is linear in the input size.

4.1. 8-Periodic sorting on the two-dimensional mesh

The first constant-periodic sorting algorithm with sublinear runtime was proposed by Schwiegelshohn [25].

Theorem 2 (Schwiegelshohn [25]). *There is an* 8*-periodic sorting network that sorts* n *items in* $O(\sqrt{n} \cdot \log n)$ *steps, where the underlying graph of the network is a mesh* (*with some wrap-around edges*).

It is astonishing how simple and simultaneously successful the construction of Schwiegelshohn is. The nodes of the network are arranged in an $N \times N$-mesh with the row-major ordering. The steps of one round for $N = 4$ are shown in Fig. 6.

The main novelty of the algorithm are the steps 4 and 8 where each time only half of the wrap-around comparators between the last and the first column are used. Such design of these wrap-around comparators is crucial in order to achieve a good runtime. Owing to them, the effect of a single tower containing 0's traveling around the network without losing its height quickly (as in Savari's approach) does not occur. This can be seen on Fig. 7 which depicts how such a tower is broken into two pieces while crossing the leftmost column.

Below we outline the analysis of Schwiegelshohn's algorithm sketched in [25] (for a detailed and complete analysis of a related 3-periodic algorithm see [13]). The analysis is based on the concept of left-, right-, up- and down-running items. Notice that comparators of a step act either separately within a column and are called *vertical*, or within

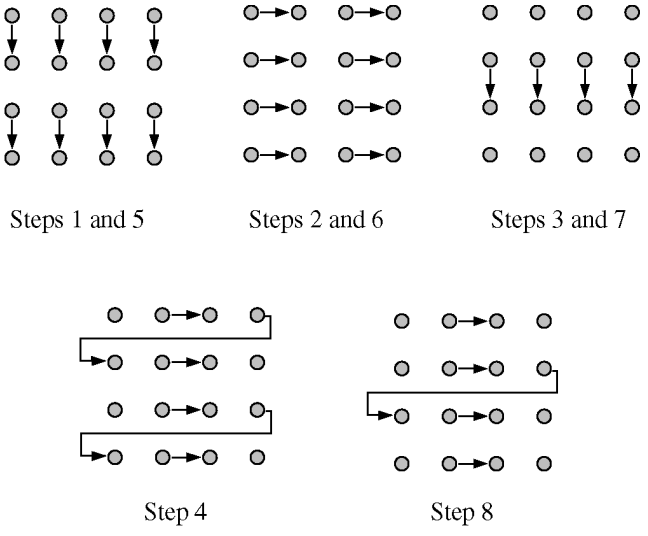


Fig. 6. The steps of one round of Schwiegelshohn's algorithm.

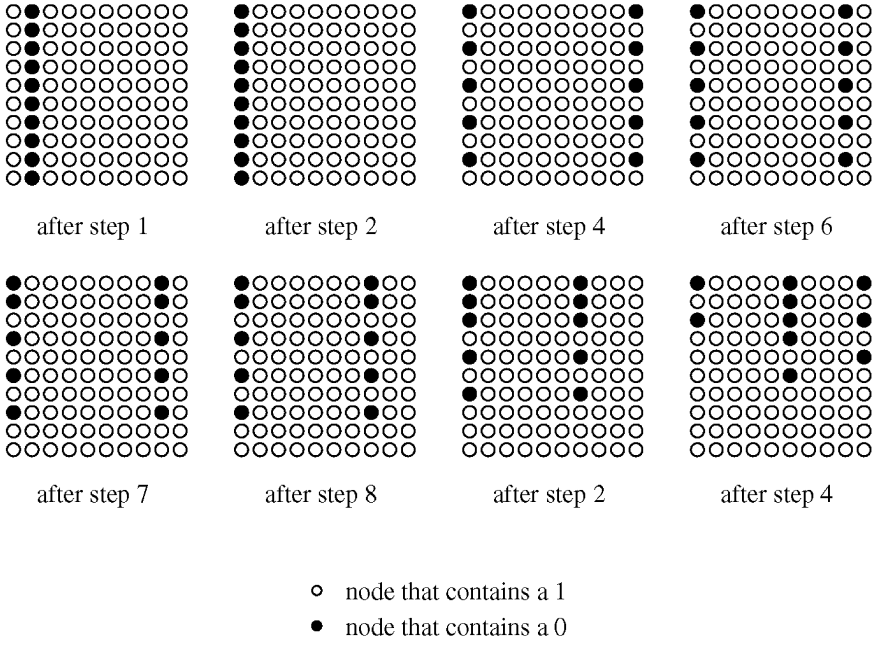


Fig. 7. Creating broken towers through wrap-around comparators.

a row and are called *horizontal*. Steps that contain mainly horizontal (vertical) comparators are called *horizontal steps* (*vertical steps*). An item α is called *right-running* (*left-running*), if during the last horizontal step α was involved in a compare-exchange operation through a horizontal comparator and placed at the right (left) end of the

comparator. A column that contains only right-running (left-running) items is called an R-*column* (L-*column*). Vertical comparators do not influence the property of being right-running (left-running). So, every column, except the first and the last one, is alternately an L-column and an R-column. We may think that the R-columns move one position to the right at each horizontal step and the L-columns move one position to the left, so we speak about *moving* columns.

One of the fundamental features of the algorithm is that consecutive steps of OETS are executed on a moving column during steps 1, 3, 5, 7. So we mimic sorting the moving columns except that R-columns may lose 1's and L-columns may loose 0's due to the *horizontal* steps 2, 4, 6, 8. Additionally, the semi-sorted state achieved may be abruptly destroyed by the wrap-around comparators. However, this is necessary to avoid the phenomenon that we have discussed for Savari's networks.

The R-columns collect 1's and L-columns lose 1's. So nearly sorted columns with many 1's move to the right border and columns with only a few 1's to the left border. Arriving at the borders such two nearly sorted columns balance their 0's and 1's by the wrap-around comparators at the steps 4 and 8. For example look what happens with a tower of zeroes arriving at the leftmost column (see Fig. 7). After a while, we see that the zeroes of the tower have been divided between the rightmost and the leftmost columns with zeroes occupying every second node up to some height in both these columns (see in Fig. 7 after Step 4). To describe this phenomenon we talk about *broken towers* generated by the wrap-around comparators.

By analyzing the movement of 0's and 1's in detail one gets that during the computation rows containing 0's only emerge at the top of the network and rows containing 1's only emerge at the bottom. The remaining area in the middle is called the *dirty area*. One can show that after $O(N)$ steps the total number of rows outside the dirty area is $\Omega(N)$. Thereby we reduce the height of the dirty region by a constant factor. Then for the dirty region we apply the same analysis. Continuing in this way $O(\log N)$ times we finally obtain a dirty region consisting of a constant number of rows. Then $O(N)$ steps suffice to finish sorting due to Schröder's Theorem (see Section 2). Thus the runtime of the algorithm is $O(N \cdot \log N)$.

4.2. 3-Periodic sorting on multi-dimensional meshes

In this section Schwiegelshohn's construction is improved by generalizing it to multi-dimensional meshes [13]. The achieved runtime is asymptotically faster than the runtime of Schwiegelshohn's construction. Further, the period is reduced to 3, which is optimal.

Theorem 3 (Kutyłowski and Loryś [13]). *Let $d \in \mathbb{N}$ be fixed. There is a* 3*-periodic comparator network that sorts n items in time $O(n^{1/d} \cdot \log^{O(d)} n)$. The underlying graph of this network is based on the d-dimensional mesh.*

Networks with such a runtime have already been presented in Section 3, but those constructions required expander-like structures and the period achieved was $\Theta(d)$.

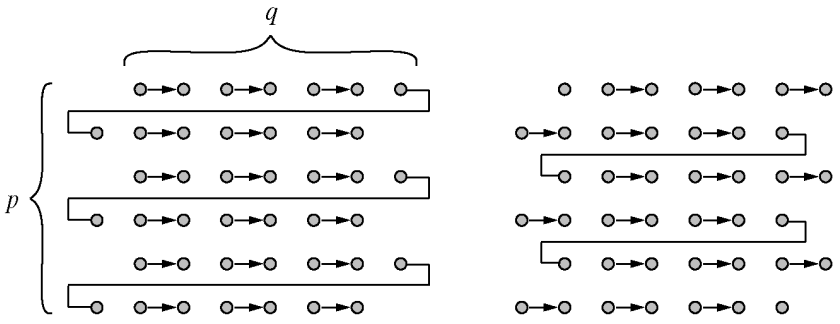


Fig. 8. The horizontal steps H_1 and H_2, respectively.

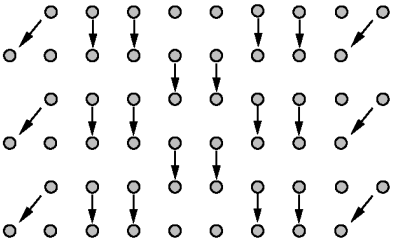


Fig. 9. The vertical step V_2 of $\mathcal{M}_2$.

The construction leading to Theorem 3 starts with a two-dimensional mesh and then is recursively generalized to higher dimensional meshes.

Two-dimensional mesh: To construct network $\mathcal{M}_2$ on the two-dimensional mesh we use nodes arranged in an $p \times q$-mesh, $q \geqslant 10 \cdot p$, where in the first and the last column of the mesh only half of the nodes are used. A round of $\mathcal{M}_2$ consists of two *horizontal steps* H_1, H_2 (see Fig. 8) and the *vertical* step V_2 (see Fig. 9).

H_1 and H_2 are in fact the two different steps of OETS. Thus the network is constructed by interleaving the genuine OETS and an additional step V_2 that contains comparators acting on columns. Similarly to Schwiegelshohn's algorithm the concepts of *moving columns*, R-*columns* and L-*columns* are used. By a horizontal step an R-column (L-column) moves one position to the right (left).

Inside all columns, except for the first two and the last two columns, V_2 performs the steps of OETS acting separately on each column. As in the case of the Schwiegelshohn's construction, the purpose of these comparators is to sort the columns (as before, the horizontal steps interfere with the progress of sorting in the moving columns). For technical reasons the comparators sorting the columns are organized in a slightly different way. Since between subsequent vertical steps a moving column is shifted two positions, a different step of OETS must be executed every second column (check how it works in Fig. 9).

The border comparators of H_1 and H_2 together with the slanted comparators of V_2 have the same purpose as the border comparators in step 4 and 8 of Schwiegelshohn's

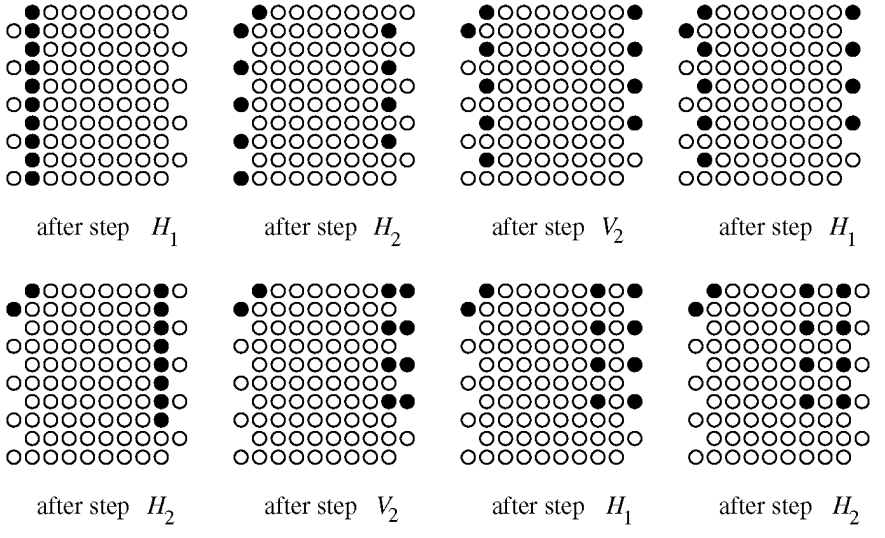


Fig. 10. Breaking towers of 0's by $\mathscr{M}_2$.

algorithm (see Section 4.1). Namely, they prevent a single tower of 0's from traveling around the network for a long time. For an example of how such a tower is broken at the border of the network inspect Fig. 10.

Now we sketch the main ideas of the runtime analysis of $\mathscr{M}_2$. First observe that if a horizontal step is executed and there are comparators between columns i and $i+1$, then afterwards the number of zeroes in column $i+1$ does not exceed the number of zeroes in column i. Simply, for each zero that is left in column $i+1$ there is a corresponding zero in the same row inside column i. It follows that an L-column S contains no fewer zeroes than any R-column which was met by S during the previous horizontal steps. Hence if the L-column S is generated at the right border and moves to the left, then the number of zeroes in S at each moment is not less than the number of zeroes in *any* R-column on the right side of S. So we see that a certain fraction of zeroes is concentrated in the L-columns after $O(q)$ steps. If we assume that the input contains at least $pq/2$ zeroes, then by counting arguments one can prove the following technical result:

> Let $s=4p+2$. There are $t=O(q)$ and $m=\Omega(p)$ such that after round t every L-column arriving at the column s contains at least m zeroes. (1)

Consider now an L-column U that arrives at the column s with at least m zeroes inside. These zeroes are affected by vertical comparators while U moves to the left up

to column $2p+2$. Since together p steps of OETS are executed on the way to column $2p+2$ and no zero may be lost by the L-column U, all zeroes mentioned have enough time to be moved to the top part of U. Therefore, we see that after $O(q)$ steps each L-column arriving at the left border contains a tower of zeroes of height at least m. Then these towers fill the top $\Omega(p)$ rows with zeroes in $O(q)$ steps as depicted by Fig. 11.

Since $\mathscr{M}_2$ is symmetric regarding the roles of zeroes and ones, an argumentation such as above shows that in $O(q)$ rounds either $\Omega(p)$ rows of 1's emerge at the bottom of the network or $\Omega(p)$ rows of zeroes emerge at the top of the network. Thereby the height of the dirty region is reduced by a constant factor. Repeating this procedure $O(\log p)$ times we reduce the size of this region to a constant. Afterwards, $O(q)$ steps suffice to finish sorting thanks just to steps H_1 and H_2 of OETS.

Multi-dimensional mesh: The main idea is to connect several copies of the network based on the $(d-1)$-dimensional mesh into a d-dimensional mesh so that the $(d-1)$-dimensional submeshes behave similarly as the columns in the two-dimensional case. Recall that for $\mathscr{M}_2$ the behavior of moving columns was crucial for the performance of $\mathscr{M}_2$: these columns move through the network and are sorted in the meantime. While arriving at the borders they contain towers of zeroes or ones, respectively. These towers are broken at the borders into two parts moved to the different sides and then sorted again.

A similar scenario holds for the d-dimensional mesh: $(d-1)$-dimensional submeshes are moved through the d-dimensional mesh, they are sorted in the meantime and then the sorted parts of the submeshes are broken at the borders and moved to the different sides of the mesh.

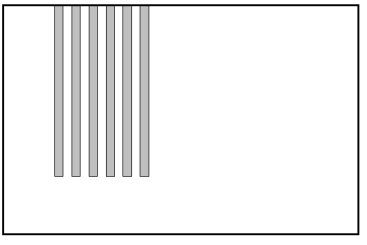
In order to give some insight in the construction of the networks, we describe the network $\mathscr{M}_3$ for dimension 3. The underlying graph of $\mathscr{M}_3$ is the three-dimensional $k_1 \times k_2 \times k_3$ mesh, where on the side walls, the front and back wall, only half of the nodes are used (see Fig. 12). The point is that if we cut such a mesh in two directions shown by Fig. 12, then we obtain submeshes of the shape used by $\mathscr{M}_2$. The nodes of $\mathscr{M}_3$ are ordered lexicographically with the first coordinate most important. A *vertical slice* s is the set of all nodes of $\mathscr{M}_3$ with the last coordinate equal to s. A *horizontal slice* t is the set of all nodes of $\mathscr{M}_3$ with the first coordinate equal to t.

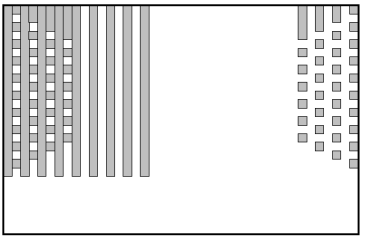
Before we define the steps executed by $\mathscr{M}_3$ we note that the mesh can be viewed in a slightly different way: By the ordering of $\mathscr{M}_3$ the nodes of the horizontal slice $i-1$ precede the nodes of the horizontal slice i, and inside a horizontal slice the nodes are with row-major ordering. It is convenient to consider the nodes of $\mathscr{M}_3$ as a two-dimensional structure by putting the horizontal slice i on top of slice $i-1$, for each i (see Fig. 13). In this way we obtain the known layout of $\mathscr{M}_2$. The ordering is row-major and a column groups the nodes of a vertical slice. This representation is called the *two-dimensional view* of $\mathscr{M}_3$.

Now, we are ready to define the steps of the algorithm. The first and second steps are the comparators of OETS according to the ordering of the mesh. If we consider the mesh in the two-dimensional view these steps are identical to the horizontal steps H_1

(a) L–columns with towers of 0's at the top move to the left border of the network.

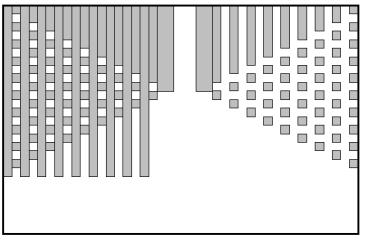
(b) L–columns with towers of 0's are broken at the left border. The 0's are divided between the leftmost and the rightmost columns. The broken columns move to the middle of the network and are sorted on the way.

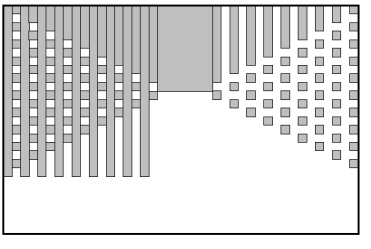




(c) The broken columns are sorted till they achieve the middle of the network.

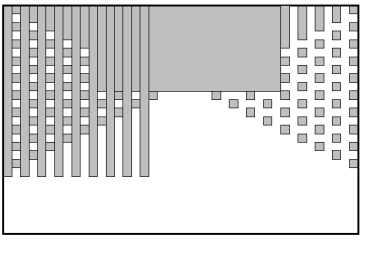
(d) There emerges an area of high towers of 0's in the middle of the network.

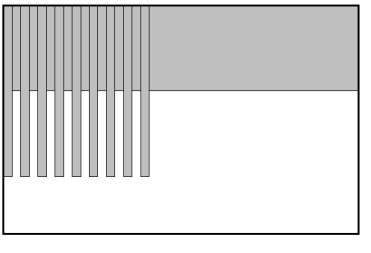




(e) The area of high towers expands to the right and left border.

(f) There are high towers of 0's in each column of the network.





node containing a 0

Fig. 11. Creating $\Omega(p)$ rows of 0's by $\mathcal{M}_2$.

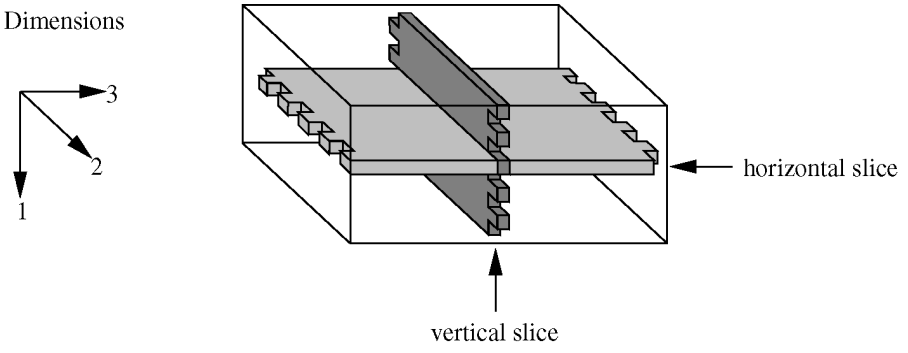


Fig. 12. The network $\mathcal{M}_3$.

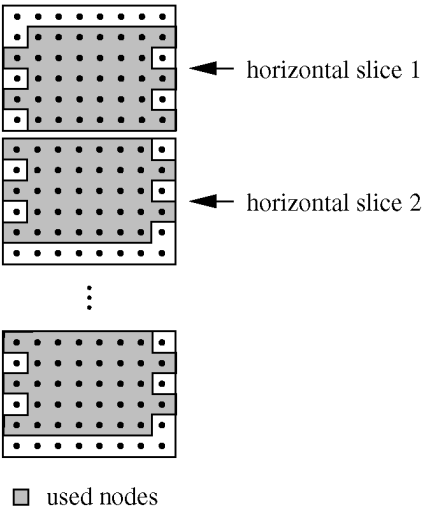


Fig. 13. The two-dimensional view of $\mathcal{M}_3$.

and H_2 (see Fig. 8). These steps are responsible for moving the vertical slices through the network. The algorithm is formed by interleaving the genuine OETS (realized by H_1 and H_2) with an additional third step V_3 that has many purposes. First, V_3 is used to sort the $(d-1)$-dimensional vertical slices and to sort the broken "towers" at the border. In the two-dimensional case both these goals have been realized by the comparators of OETS in the columns. Here we use $\mathcal{M}_2$ to sort the submeshes. Now, we get into troubles, since the two-dimensional algorithm does not sort the broken vertical slices in the way we wish. Therefore we have to arrange comparators in the vertical slices on the right and left border of the network in the way that handles sorting of broken vertical slices. Due to all these goals we compose V_3 from comparators defined separately for the following groups of vertical slides, called B_L (B_R), A_L (A_R), and S (see Fig. 14):

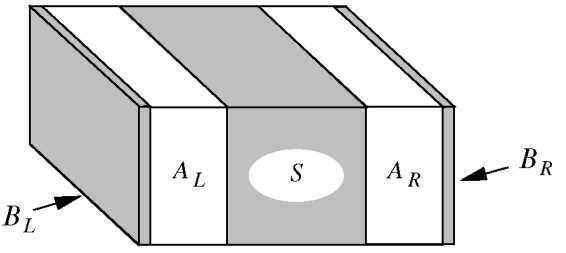


Fig. 14. Partitioning of the vertical slices of V_3.

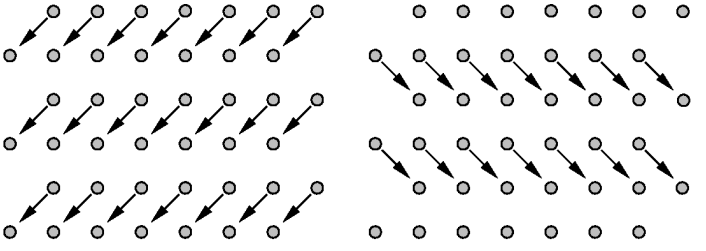


Fig. 15. The two vertical slices of $A_{\mathrm{L},1}$ and $A_{\mathrm{R},1}$.

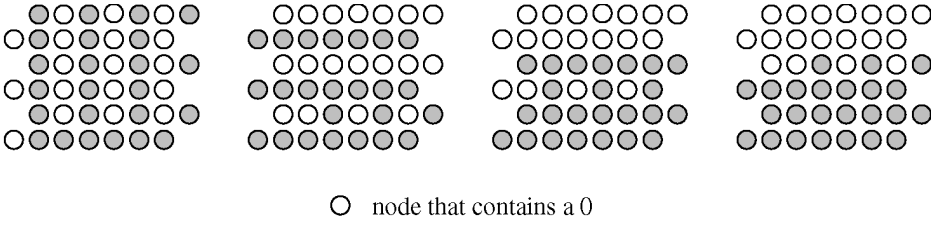


Fig. 16. Sorting of broken towers by the two vertical slices of $A_{\mathrm{L},1}$ and $A_{\mathrm{R},1}$.

B_{L} (B_{R}) consists of the two leftmost (rightmost) vertical slices. They contain the slanted edges similar to the slanted edges in V_2 (see Fig. 8). The comparators have the same purpose as slanted edges for the two-dimensional case. Namely they are for breaking "towers".

A_{L} and A_{R} are concatenations of the two groups $A_{\mathrm{L},1}$ $|A_{\mathrm{L},2}$ and $A_{\mathrm{R},2}|$ $A_{\mathrm{R},1}$, respectively.

$A_{\mathrm{L},1}$ ($A_{\mathrm{R},1}$) consists of k_1 vertical slices next to B_{L} (B_{R}). The comparators in $A_{\mathrm{L},1}$ ($A_{\mathrm{R},1}$) are used to sort the broken towers. They connect nodes in different rows of a vertical slice as shown by Fig. 15. The choice of the different vertical slices is done so that moving L-slices or R-slices are affected alternately by these two slices. By the comparators in $A_{\mathrm{L},1}$ ($A_{\mathrm{R},1}$) the broken towers are sorted together row by row (see Fig. 16).

$A_{L,2}$ ($A_{R,2}$) consist of k_2 vertical slices. If we consider $\mathcal{M}_3$ in the two-dimensional view the columns contain the two different steps of OETS. The choice of the two different steps is done so that moving L-slices or R-slices are affected alternately by these two steps. These steps of OETS sort the part of the broken towers that will not be sorted by the comparators in $A_{L,1}$ ($A_{R,1}$) (see Fig. 16). Note that the comparators in $A_{L,i}$ ($A_{R,i}$) connect nodes in dimension i.

The region S is used to sort moving vertical slices. S consists of the remaining vertical slices in the middle of $\mathcal{M}_3$. Each of this slices implements one of the steps H_1, H_2 and V_2 of $\mathcal{M}_2$. The choice of the step to be implemented on a given vertical slice is met so that every moving L-slice (R-slice) is affected repeatedly by the consecutive steps of $\mathcal{M}_2$. Such an arrangement is quite easy to obtain. Note that the different steps of $\mathcal{M}_2$ have to be embedded in both directions. Moving L-slice (R-slice) will be sorted on the way through S. Therefore S consists of $r \geqslant ck_2 \log k_1$ vertical slices, for sufficiently large c. In order to apply counting arguments S consists of $\Theta(k_2 \log k_1)$ vertical slices.

Analogously to Claim (1) on p. 12 one can prove the following claim:

> Let $s = 2 + k_1 + k_2 + ck_2 \log k_1$. There are $t = O(k_3)$ and $m = \Omega(k_1 k_2)$ such that after round t every L-column arriving at the column s contains at least m zeroes. (2)

The runtime analysis of the network constructed follows the same sequence of arguments as this for the two-dimensional case. An L-column that arrives at the column s with at least m zeroes inside is affected by vertical comparators while moving to the left through S. All zeroes have enough time to move to the top part of a column. Therefore, after $O(k_3)$ steps each L-column arriving at the left border contains a tower of 0's of height at least m. The analysis corresponding to Fig. 11 is technically a little bit different due to the fact that we have groups A_L and A_R for sorting broken towers and for expanding high towers of 0's to the right and left border of the network. Thus after $O(k_3)$ steps either $\Omega(k_1 k_2)$ top rows contain only 0's or $\Omega(k_1 k_2)$ bottom rows contain only 1's. It follows that the total time for sorting is $O(k_3 \log(k_1 k_2))$. For $k_3 = \Theta(n^{1/3} \log^{2/3} n)$, $k_1, k_2 = \Theta(n^{1/3} / \log^{1/3} n)$, we get the total runtime of $O(n^{1/3} \log^{5/3} n)$ steps.

The construction of $\mathcal{M}_d$ for dimension d is similar to the construction of $\mathcal{M}_3$, since one can prove by induction a two-dimensional view for $\mathcal{M}_{d-1}$. Corresponding to $\mathcal{M}_3$ the group A_L (A_R) is a concatenation of $d-1$ subgroups $A_{L,i}$ ($A_{R,i}$) for $i = 1$ to $d-1$. The comparators in the subgroups $A_{L,i}$ ($A_{R,i}$) connect nodes in dimension i for $i = 1$ to $d-1$.

5. Periodification scheme

The main step in the development of constant-periodic networks is the construction of a general method called *the periodification scheme*. It is a method converting an

arbitrary sorting network into a constant-periodic sorting network with at most slight loss of efficiency. A construction yielding period 5 is presented in [13] (for a full version see [14]). An improved construction giving a 3-periodic network can be found in [18].

Theorem 4 (Kutyłowski and Loryś [13] and Oesterdiekhoff [18]). *For a standard (non-periodic) comparator network that sorts n items in time $T(n)$, and there exists a 3-periodic comparator network that sorts $\Theta(nT(n))$ items in time $O(T(n)\log n)$.*

The periodification scheme is constructive and does not generate additional big overhead. Hence by applying it to easy networks, we get easy networks. In particular, applying the periodification scheme to Batcher's algorithms, we get 3-periodic comparator networks that sort n items in time $O(\log^3 n)$. To achieve asymptotically better runtime, one may use the complex AKS network to construct a 3-periodic comparator network that sorts n items in time $O(\log^2 n)$. Finally, note that for certain functions T (for instance, $T = n^\varepsilon$), the above transformation even improves the runtime compared to the original network.

The periodification scheme transforms sorting networks into another sorting networks preserving their relevant properties and extending them by an additional important property (periodicity). Another construction of this kind (Columnsort) has been proposed by Leighton [17]. His procedure transforms an arbitrary sorting network into a sorting algorithm running on a certain processor network of degree 3.

Idea: Considering Schwiegelshohn's algorithm and the related 3-periodic algorithm $\mathscr{M}_2$ we observe that in each column the steps of OETS are performed. We have already seen while constructing $\mathscr{M}_d$ that through replacing OETS by the algorithm designed for two-dimensional meshes we may speed up the algorithm. Here we go beyond the results that may be obtained through multiple application of mesh algorithms. Consecutive columns may perform the consecutive steps of an *arbitrary* sorting algorithm, maybe a fast one. In this way we may reduce the number of "columns" we need.

This simple idea can be implemented, the proof goes similarly as in the two-dimensional case, however on certain places we get technical problems. One of them, as for the meshes, is handling broken towers on the borders.

Construction of the network: Let N be a standard (non-periodic) comparator network that sorts n items in $T(n)$ steps. We convert N into a constant-periodic network $\mathscr{P}_N$. The nodes of $\mathscr{P}_N$ are arranged as nodes of a mesh, with n rows and q columns, $q \geqslant b \cdot T(n)$ for a sufficiently large constant b, corresponding to the case of $\mathscr{M}_2$. The underlying graph of $\mathscr{P}_N$ contains edges inside the rows and wrap-around edges just like in the case of $\mathscr{M}_2$. Only the edges connecting the nodes inside the columns are different: they might be long (so they are no longer the edges of a mesh) and correspond in some way to the comparators of N.

A round of $\mathscr{P}_N$ consists of two *horizontal* steps H_1, H_2 (see Fig. 8) and a *vertical* step V. Steps H_1, H_2 are the steps of OETS. The additional step V has a similar purpose to the vertical step for the networks based on multi-dimensional meshes.

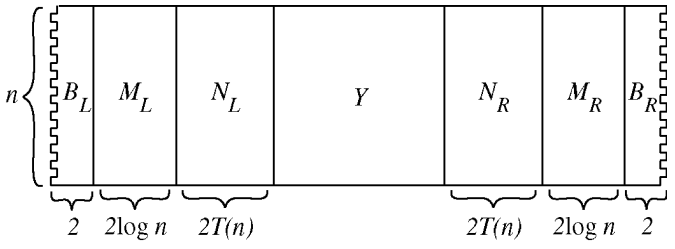


Fig. 17. Partitioning of the columns of $\mathscr{P}_N$ for V.

As for the construction of $\mathscr{M}_d$ different columns in $\mathscr{P}_N$ have different purposes during the computation on $\mathscr{P}_N$. Therefore, we partition the columns of $\mathscr{P}_N$ into groups B_L, M_L, N_L, Y, N_R, M_R and B_R according to their different purposes (see Fig. 17).

B_L and B_R contain the slanted edges (see Fig. 9), as for the algorithms previously described. They break towers of 0's or 1's arriving at the borders.

The groups M_L (M_R) are used to sort the broken columns. We have already solved this problem for the multi-dimensional meshes, but now we may require that M_L and M_R consist of much less columns than before. This is necessary in order to keep the total number of columns $\mathrm{O}(T(n))$. Therefore, we apply a solution that requires only $\mathrm{O}(\log n)$ columns. For this purpose consider the network S depicted in Fig. 18. Network S is essentially the Odd–Even Merger described by Batcher in [2], but the first step is omitted. S is in fact Odd–Even Merger if one input sequence is stored by the even numbered nodes and the other by the odd numbered nodes. This is what we need, since a broken tower consists of zeroes on even (odd) positions and ones on odd (even) positions. Thereby, roughly speaking, we have two sorted subsequences as required by the merger. In order to sort the broken towers, we *embed* S into $\mathscr{P}_N$. More precisely, the ith layer of S is implemented by column $2i-1$ of M_L counting from the left and by column $2i-1$ of M_R counting from the right. Therefore, an R-column (L-column) that is shifted through M_L (M_R) is affected by consecutive steps of S.

The groups N_L and N_R are used to sort, respectively, L- and R-columns. For this purpose we *embed* N into each of these groups. This means that layer i is implemented by column $2i-1$ of the group so that a moving column shifted through the group is affected by consecutive steps of N.

Group Y consists of $\Theta(T(n))$ columns in the middle of $\mathscr{P}_N$. The purpose of Y is only to increase the width of $\mathscr{P}_N$. This is necessary in order to apply counting arguments while proving the following claim analogous to Claim (1) on p. 12.

> Let $s = 2\log n + 2T(n) + 2$. There are $t = \mathrm{O}(T(n))$ and $m = \Omega(n)$ such that after round t every L-column arriving at the column s contains at least m zeroes. (3)

Concerning the runtime analysis a similar sequence of arguments applies as for the algorithms described previously. An L-column that arrives at the column s with at least

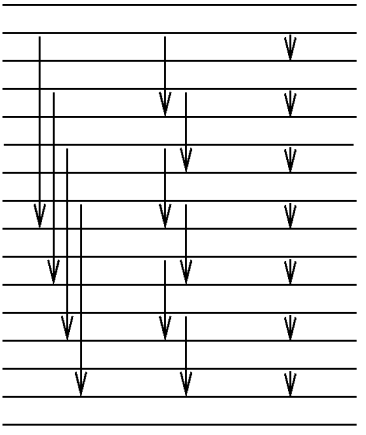


Fig. 18. Network S for $p = 16$.

m zeroes inside is affected by vertical comparators while moving to the left through N_L. Therefore after $O(T(n))$ steps each L-column arriving at the left border contains a tower of zeroes of height at least m. The analysis corresponding to Fig. 11 differs a little bit from the algorithms previously described due to the fact that there are the groups M_L and M_R for sorting the broken towers and for extending the high towers of 0's to the left and right border. Thus finally after $O(T(n))$ steps of $\mathscr{P}_N$ either $\Omega(n)$ top rows contain only 0's or $\Omega(n)$ bottom rows contain only 1's. It follows that after $O(T(n)\log n)$ steps the input consisting of $\Theta(nT(n))$ items becomes sorted.

6. Merging

In this section we consider the problem of merging two sorted sequences. The classical solution of this problem is given by Batcher's Odd–Even Merge and Bitonic Merge networks running in $\log n$ time. In this section we describe how to merge such sequences in time $O(\log n)$ on constant-periodic comparator networks. Kutyłowski et al. [12] (for a short version see [11]) prove that there is a 3-periodic comparator network that merges two sorted sequences of $n/2$ numbers in time $12\log n$. By increasing the period (but still keeping it constant) and fine tuning of the network the runtime can be decreased to approximately $2.25\log n$. Note that the runtimes achieved differ only by a small constant factor from the lower bound $\log n$, although the restriction of periodicity is very strong. Moreover, the constant-periodic networks constructed have very simple architecture.

In order to give some insight into the construction of the merging networks, we sketch the proof of the following theorem:

Theorem 5 (Kutyłowski [11]). *There is a constant-periodic comparator network M that merges two sorted sequences of $n/2$ items in time* $O(\log n)$.

General structure of the algorithm: The nodes of network M are arranged in a $p \times q$-rectangle with $q = O(\log p)$. Let $P_{i,j}$ denote the node in the row i and the column j, and let C_j denote the jth column of M. The nodes of M are in *snake-like* order. (Note the difference to constant-period sorting networks where we have used row-major ordering!)

The algorithm has a notable property that at each step of the computation if an element moves from column C_i to C_j, then *every* element stored in C_i moves to C_j. This property holds only for the input sequences consisting of 0's and 1's, and under the interpretation that the comparators may switch equal elements. (Thereby our analysis collapses if we do not use the 0–1-Principle.)

Due to the effect of "exchanging places" by the columns we may talk again about *moving columns* that are shifted around the network. The purpose of these movements is that at different positions different sets of comparators are applied inside a column. These comparators are used to sort the contents of the moving columns.

The input allocation to this network has the property that the number of 1's at each column is the same up to one. Since the columns do not mix, this property will be preserved all the time. Hence after sorting every moving column, the contents of the network is sorted except for at most one row. Then some additional rounds suffice to sort this row.

After the simple outline of the algorithm we discuss below some technical details that are crucial for its implementation.

Input allocation and basic properties: We use the following easy input allocation: The elements of the first (second) sorted sequence are loaded into the odd (even) rows; the elements of both input sequences are placed according to the snake-like ordering. So it is easy to see that at the beginning of the computation the following properties hold:

1. the number of 1's in the columns of M may differ by at most 1;
2. the contents of each column has the form $1^d(01)^e0^*$ for some e and d, that is, d bottom positions are occupied by 1's, the $p-2e-d$ top positions are occupied by 0's, and in the remaining $2e$ positions there are alternately 0's and 1's.

One of the main features of the algorithm is that the properties mentioned above hold during the *whole computation*.

The algorithm performs one or more *vertical* steps and two *horizontal* steps that exchange elements between consecutive columns in the same row according to the snake-like ordering (see Fig. 19). The purpose of horizontal steps is to shift the moving columns around the network. Vertical steps are used for sorting the moving columns (so called *jump steps*). Some networks use auxiliary vertical steps for keeping the columns in a right form for horizontal moves.

Exchange of columns: As we have already said, the main feature of the algorithm is that during a horizontal step two columns exchange their places. This phenomenon holds provided that the number of ones in the columns involved differ by at most one and they have both the forms $1^d(01)^e0^*$ and $1^{d'}(01)^{e'}0^*$ for some e,d and e',d'.

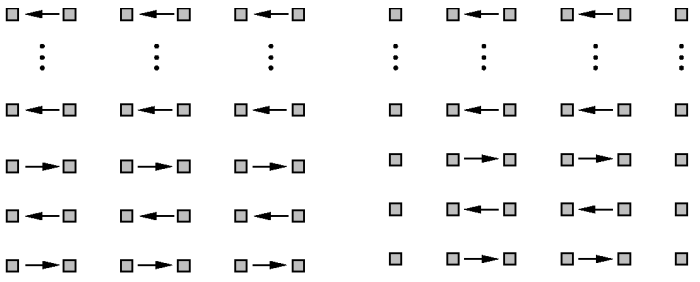


Fig. 19. The horizontal steps.

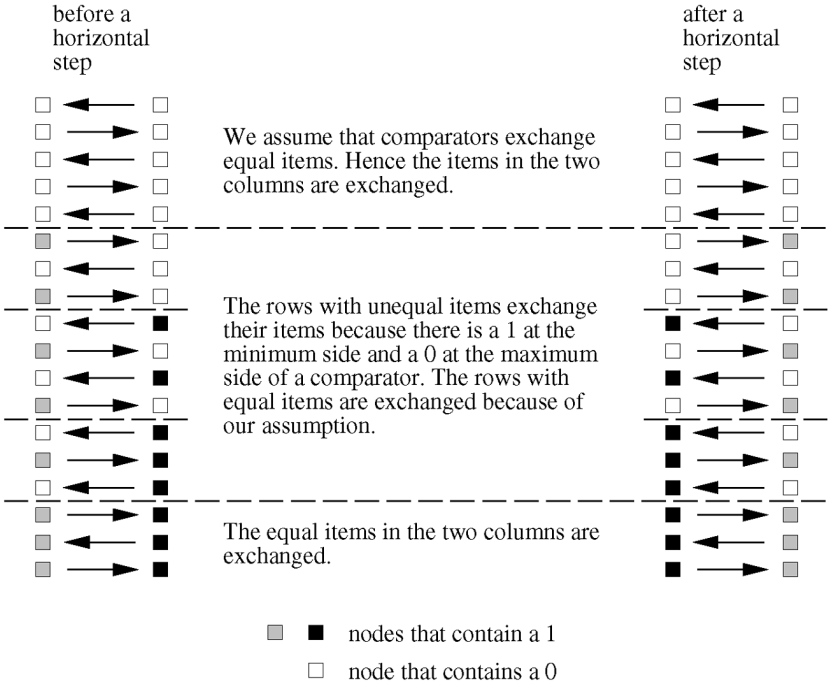


Fig. 20. Exchanging the contents by the neighboring columns during a horizontal step.

A column with the contents of the form $1^d(01)^e0^*$ (from the bottom to the top) is classified as an *E-column* (O-*column*) if d is even (odd). The key to our construction is that if the properties mentioned hold, C_i is an E-column, C_{i+1} is an O-column and a horizontal step has comparators between columns C_i and C_{i+1}, then C_i and C_{i+1} exchange their contents. How it works can be seen in an example given by Fig. 20.

Fullfiling the assumption that a column is an E- or O-column may be guaranteed by an additional *ordering* step consisting of appropriate OETS steps executed on the columns (for instance in order to guarantee that column C_i becomes an E-column even if it has been an O-column is done by applying comparators $(P_{i,j}, P_{i,j+1})$ for j even).

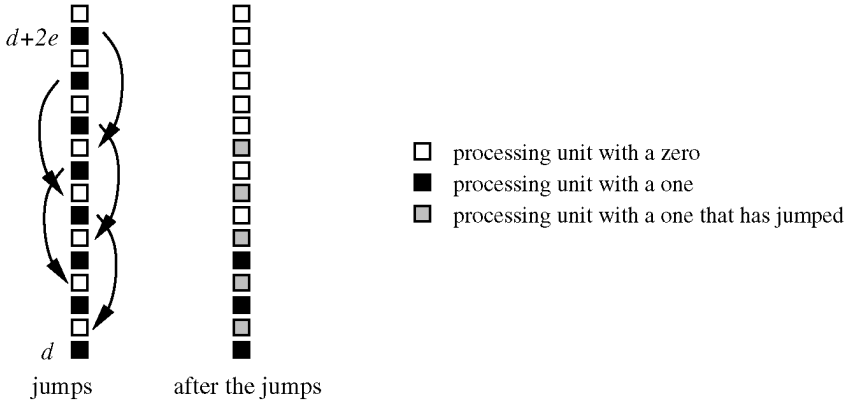


Fig. 21. A jump column for the case $2e = 3\ell - 1$.

Jumps: Now we describe jump steps responsible for sorting the columns. Such a step executes *jumps* separately in each column. A *jump* of size k inside column C_i consists of *jump comparators* $(P_{i,j}, P_{i,j+k})$, for all even j or for all odd j. We assume that the jump size must be odd, so all comparators of a jump originate in odd (even) rows and point to even (odd) rows.

Recall that during the computation each column has the form $1^d(01)^e0^*$. By a 01-*region* of this column we mean the part of the column consisting of positions $d+1$ through $d+2e$ containing alternately zeroes and ones. The jumps are designed so that they originate in positions where we may expect to find 1's of the 01-region. The idea is that with some luck some of the 1's of the 01-region jump into the places at the bottom of the 01-region occupied previously by 0's and thereby reduce the size of the 01-region (see Fig. 21). Of course, the effect depends on the relative size of the 01-region and the jump size: Suppose that a column C_i has the form $1^d(01)^e0^*$ and let ℓ be the jump size at column C_i. Then a lucky situation is for example when the size of the 01-region is bounded by $2e \leqslant 3\ell - 1$ (see Fig. 21). Then the bound on the size of the 01-region is reduced to $2(e-\ell) \leqslant \ell - 1$, that is, to about one third of the original value. Thus applying $\log_3 p$ jumps of sizes $p/3$, $p/3^2$, $p/3^3, \ldots$, on a moving column reduces the size of its 01-region to 0.

We arrange the jumps inside the columns so that if a moving column starts at some fixed point of the network, then consecutive jumps reducing the size of its 01-region will be applied during the next vertical steps. For each moving column sorting consists of two conceptual phases. First the moving column is shifted through the network until it reaches the starting point mentioned above. During the first phase the size of the 01-region of this column may be somewhat reduced, but we do not count on that. The first phase takes $O(q)$ steps. During the second phase the size of the 01-region is guaranteed to be reduced to 0. Again it takes $O(\log p) = O(q)$ steps. Finally, when every moving column is already sorted, then there is at most one row containing zeroes and ones that still need to be sorted. Now the only observable activity of M is sorting

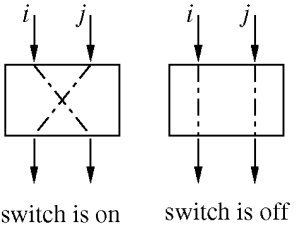


Fig. 22. Different settings of a switch.

this row by OETS (the appropriate comparators are provided by the horizontal steps). Therefore in $O(q)$ additional steps this last row becomes sorted. Since $q = O(\log p)$ the runtime of the algorithm is bounded by $O(\log n)$ as stated by Theorem 5.

7. Permutation networks

In this section we refer to a method related to the periodification scheme that has been applied to construct periodic switching networks for certain classes of permutations [8]. We consider the problem of permutation routing: sending n packets from nodes $o_1, \ldots, o_n$ to destination nodes $d_1, \ldots d_n$, where the packet from o_i goes to node $d_{\pi(i)}$ and π is a permutation over $\{1, \ldots, n\}$. The simplest networks realizing permutation routing are switching networks [10, p. 243]. We may describe them in the following way: There are n wires leading from o_i to d_i, $i \leqslant n$. The wires are connected by switches organized in some d layers. Within a layer one wire may be connected to at most one switch. A switch connects exactly two wires and works as follows: if a switch connecting wires i and j is off, then the packets traveling along wires i and j are unaffected by the switch. If it is on, then the packet from wire i goes to wire j, and the packet from wire j goes to wire i (see Fig. 22). Note the similarity to comparator networks!

The quality of a switching network can be measured by the total number of switches, the number of layers, and the number of permutations that can be realized by the network. We will focus our attention on constant period switching networks: a switching network of d layers has period c, if for every $i \leqslant d-c$, the layers i and $i+c$ are identical (nevertheless, the setting of switches might be different!).

By combining known results it can be shown that all permutations of n elements can be realized by a 3-periodic switching network of $O(\log n)$ parallel layers [26]: the key point is that any permutation can be routed on the shuffle–exchange network in $4 \log n$ steps (see e.g. [16, p. 493]). The cyclic shifts of the shuffle–exchange network can be performed by two layers (see Fig. 23) and one additional layer is needed for the exchange edges. (The left-shift can be realized by performing twice the two layers of the right-shift, where the first and the last of the four layers is off.)

Often periodic switching networks for certain classes of permutations are constructed. An example of such a class important for applications is the set of all cyclic shifts

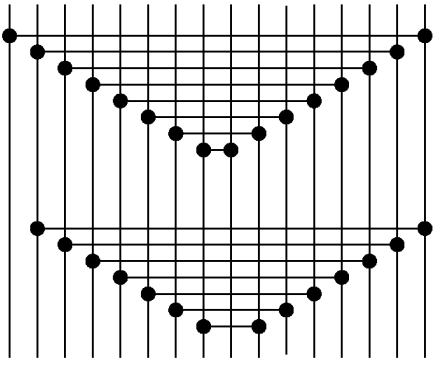


Fig. 23. A switching network of two layers realizing a cyclic shift of length 1.

$sh_a(x) = x + a \bmod n$, for all $a \in \{2^i \mid i \leqslant \lfloor \log n \rfloor\}$ [7]. Since the number of shifts in this class is small, a lower bound of $\Omega(\log n)$ on the number of layers of switching networks realizing all permutations of n elements does not apply. In fact, a switching network of $O(\log \log n)$ layers has been designed for realizing all these shifts [7]. Now, we sketch a "periodification construction" given by Kanarek [8] that yields the following result:

Theorem 6 (Kanarek [8]). *There is a constant-periodic switching network of* $O(\log \log n)$ *layers that can be used to realize each cyclic shift* sh_a *for* $a \in \{2^i \mid i \leqslant \lfloor \log n \rfloor\}$.

For technical reasons the network constructed by Theorem 6 consists of two subnetworks that are interleaved. Depending on which shift we are to realize, we deactivate one of these subnetworks, that is, set all its switches off. The subnetworks constructed as *parallel embedding* and *periodic linear network* are described below.

Parallel embedding: The goal of this network is to realize small shifts. However, since the shift size is always a power of 2, most shifts considered fall into this category. For an arbitrary N let $\mathcal{N}_N$ be a (non-periodic) switching network of $d = O(\log \log N)$ layers for the cyclic shifts by powers of 2 [7]. Consider N such that $Nd(N) = n$. (In fact, it would suffice to take the largest N such that $Nd(N) \leqslant n$, for the sake of simplicity we assume that the equality holds.) Let $L_1, \ldots, L_d$ be the different layers of $\mathcal{N}_N$. We *embed in parallel* the network $\mathcal{N}_N$ into a network of period 3 on n nodes:

(a) The first layer consists of the copies of $L_1, \ldots, L_d$. L_1 is applied to the first N input nodes, L_2 to the next N nodes, and so on.

(b) Let G_j be the group of nodes i with $j = i \bmod N$.The second and the third layer realize a single shift of length N, i.e. they perform shifts by length 1 inside the groups G_j.

For part (b) we have to recall that two layers suffice to realize a fixed but arbitrary cyclic shift (see Fig. 23 for a network realizing shift of length 1).

Our periodic network consists of $O(d)$ layers: Layer i is identical to the layer i mod 3 described above. The cyclic shifts sh_a for $a \leqslant N$ may be realized as follows. Let W_i

denote the nodes $(i-1)\cdot N+1,\ldots,i\cdot N$ and P_i the packets that are input to W_i. If $i>1$, then within the first 3 layers the packets P_i are shifted by N positions (only the second and the third layer are activated for them). Then they are shifted by the next N positions, and so on, till the packets P_i are moved to W_1. Then from the next three layers we use the first layer (so perform L_1) and the second and the third (the packets of P_i, already influenced by L_1, are moved to W_2). Then within the next three layers we apply L_2 and shift cyclically the results by N. The procedure continues until we apply L_d. Then we make only cyclic shifts by N until the P_i are back within the nodes of W_i. At this moment all packets are already shifted cyclically by a positions, but *within* W_i. Thereby some packets are at wrong places, namely those that instead of leaving W_i came back to the beginning of W_i. There are a such packets in each W_i. So finally we activate the first a groups G_j of (b). They shift each packet from node $i\cdot N+j$, $i\leqslant n/N$, $j\leqslant a$ to node $(i+1)N+j \bmod n$. After this operation all packets are at the right places.

Periodic linear network: The parallel embedding technique can be used for all but a few long cyclic shifts sh_{2^i}, $i\leqslant \log n$. For these remaining r shifts $q_1, q_2,\ldots,q_r$ we use another technique. There is a 6-periodic switching network of $O(r)$ layers that realizes all shifts sh_t for $t\in\{q_1,\ldots,q_r\}$.

In order to concentrate on the main issue we assume that r divides n. Then we partition the nodes into r groups, in the group G_i there are all nodes j with $i=j \bmod r$. A period of our switching network consists of:

(a) two layers that perform shifts separately in each group G_i, for G_i it is $sh_{\lfloor q_i/r\rfloor}$,

(b) two layers that perform sh_1 within each group of nodes $W_i=\{(i-1)r+1,\ldots,i\cdot r\}$,

(c) two layers that perform sh_1 on all nodes.

The computation consists now of three (conceptual) phases for each packet. If we are realizing q_i, then the jth packet p_j, $j=r\cdot(s-1)+u$, is moved as follows:

Start phase: p_j is moved through cyclic shifts inside W_s to the ith position in W_s.

Approximation phase: p_j is shifted by $\lfloor q_i/r\rfloor$ positions inside G_i using the layers of the first group.

Synchronization phase: p_j is moved to position u inside group $W_{s+\lfloor q_i/r\rfloor}$.

After the three phases all packets are moved by $\lfloor q_i/r\rfloor r$ positions. What remains to be done is to perform sh_1 on all nodes $q_i-\lfloor q_i/r\rfloor r$ times, which is less than r times.

8. Conclusions

We have presented several comparator network algorithms that have a constant period. It was even possible to obtain period 3, which is optimal for sublinear time algorithms. Runtimes obtained are astonishingly close to those obtained by non-periodic algorithms.

A challenging problem that remains open in this area is whether it is possible to design a comparator network of a constant period that sorts n numbers in time $o(\log^2 n)$.

So far no lower bound is known that would distinguish between sorting using periodic and non-periodic comparator networks.

Acknowledgements

I would like to thank Mirosław Kutyłowski for a lot of helpful discussions to finish this paper.

References

[1] M. Ajtai, J. Komlós, E. Szeméredi, Sorting in $c \cdot \log n$ parallel steps, Combinatorica 3 (1983) 1–19.
[2] K.E. Batcher, Sorting networks and their applications, in AFIPS Conference Procedings 32, 1968, pp. 307–314.
[3] R.I. Becker, D. Nassimi, Y. Perl, The new class of g-chain periodic sorters, in Proceeding 5th ACM-SPAA, 1993, pp. 356–364.
[4] V. Chvátal, Lecture notes on the new AKS sorting network, Technical Report DCS-TR-294, Rutgers University, Computer Science Department, November 1992.
[5] M. Dowd, Y. Perl, L. Rudolph, M. Saks, The periodic balanced sorting network, J. ACM 36 (1989) 738–757.
[6] D.M. Gordon, Parallel sorting on Cayley graphs, Algorithmica 6 (1991) 554–564.
[7] J. Hromkovič, K. Loryś, P. Kanarek, R. Klasing, W. Unger, H. Wagener, On the sizes of permutation networks and consequences for efficient simulation of hypercube algorithms on bounded-degree neworks, in Proceedings 12th STACS, 1995, pp. 255–266.
[8] P. Kanarek, Realization of permutations via bounded-degree networks, Ph.D. thesis, Institute of Mathematics, University of Wrocław, 1997.
[9] M. Kik, M. Kutyłowski, G. Stachowiak, Periodic constant depth sorting networks, in Proceedings 11th STACS, 1994, pp. 201–212.
[10] D.E. Knuth, The Art of Computer Programming, Vol. 3, Sorting and Searching, Addison-Wesley, Reading, MA, 1973.
[11] M. Kutyłowski, K. Loryś, B. Oesterdiekhoff, Periodic merging networks, in Proceedings 7th ISAAC, 1996, pp. 336–345.
[12] M. Kutyłowski, K. Loryś, B. Oesterdiekhoff, Periodic merging networks, Technical Report TR-rsfb-96-032, Universität-GH Paderborn, Heinz-Nixdorf-Institut, 1997.
[13] M. Kutyłowski, K. Loryś, B. Oesterdiekhoff, R. Wanka, Fast and feasible periodic sorting networks of constant depth, in Proceedings 35th IEEE-FOCS, 1994, pp. 369–380.
[14] M. Kutyłowski, K. Loryś, B. Oesterdiekhoff, R. Wanka, Constructing sorting networks with constant period, Technical Report TR-RF-95-009, Universität-GH Paderborn, Heinz-Nixdorf-Institut, 1995.
[15] M. Kutyłowski, R. Wanka, Playing tetris on meshes and multi-dimensional shearsort, in Proceedings 8th ISAAC, 1997.
[16] F. T. Leighton, Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes. Morgan Kaufmann, Los AHos, CA, 1992.
[17] T. Leighton, Tight bounds on the complexity of parallel sorting, IEEE Trans. Comput. 34 (1985) 344–354.
[18] B. Oesterdiekhoff, On the minimal period of fast periodic sorting networks, Technical Report TR-RI-95-167, Universität-GH Paderborn, Heinz-Nixdorf-Institut, 1995.
[19] M.S. Paterson, Improved sorting networks with $O(\log n)$ depth Algorithmica 5 1990 75–92
[20] L. Rudolph, A robust sorting network, IEEE Trans. Comput. 34 (1985) 326–335.
[21] K. Sado, Y. Igarashi, Some parallel sorts on a mesh-connected processor array and their time efficiency, J. Parallel Distributed Comput. 3 (1986) 398–410.
[22] S.A. Savari, Average case analysis of five two-dimensional bubble sorting algorithms, in Proceedings 5th ACM-SPAA, 1993, pp. 336–345.

[23] I.D. Scherson, S. Sen, A. Shamir, Shear-sort: a true two-dimensional sorting technique for VLSI networks, in Proceedings IEEE Int Conf on Parallel Processing, 1986, pp. 903–908.
[24] H. Schröder, Partition sorts for VLSI, in I. Kupka (Ed.), Proceedings 13th GI-Jahrestagung, Vol. 73 of Informatikfachberichte, Springer-Verlag, Berlin, 1983, pp. 101–116.
[25] U. Schwiegelshohn, A shortperiodic two-dimensional systolic sorting algorithm, in International Conference on Systolic Arrays, 1988, pp. 257–264.
[26] R. Wanka, Paralleles Sortieren auf mehrdimensionalen Gittern, Dissertation, Universität-GH Paderborn, 1995.